The Future of OOP

By Bjarne Stroustrup and Tim Lindholm

Betting on Java? Sticking with C++? Considering C#? Check out what these experts have to say about where OOP languages are and where they're going.

With rapid advancement in technology and ever-shrinking product cycles, many developers simply don't have time to keep up with more than one language. The choice they make can determine the types of projects they work on, the companies they work for, even whether their skills are considered valuable or obsolete. To help sort out the relevance—both present and future—of the two most popular OOP languages, C++ and Java, we talked with Bjarne Stroustrup, the designer of C++, and Tim Lindholm, distinguished engineer at Sun Microsystems. Here are their views on the future of programming, competing languages, evolution vs. revolution of OOP languages, and programming standards.

The Future
What will change in the way developers write code in the next three years?

Stroustrup: In C++, without appropriate libraries, anything significant seems complicated. With appropriate libraries, just about anything becomes manageable. Building and using libraries will become increasingly important. This implies an increase in generic programming, because only through that can libraries become general enough and efficient enough. I also expect to see growth in distributed computing and in the use of "components." To most programmers, these developments will manifest themselves through the use of libraries that provide convenient access to such systems.

Lindholm: Two driving forces for developers who write code will continue to be networking and distribution—the need to write applications that are not designed for use by a single computer. More applications will be written not as standalone applications tied to one device, but rather as platform-independent, distributed applications, with enabling technologies such as EJB and JSP. Programmers will have to confront Deutsch's Seven Fallacies of distributed computing. This will challenge the design patterns on which many programmers rely, as well as their skills and intuition. This is transcendent of the language choice, though design features of the language might help or hinder this shift.
A component of the growth of networking is the complementary proliferation of small, mobile devices and specialized Internet "appliances," each with its own operating system, or with operating systems common only within that particular device's space. We can now list those device spaces—home appliances, cell phones, pagers, PDAs, auto networking—but the day is quickly coming when the range and depth of these devices will be enormous. We all know this market is huge, beyond anything we imagined with the rise of the PC. Competition for the application space on these devices will be wicked, and one of the keys to winning will be fast time-to-market. Developers will need tools that enable them to write and debug their software quickly and efficiently. Platform independence will also be key in enabling programmers to develop to and support multiple device platforms.

One other change I foresee is the increased ability to better develop applications that build synergy between code (Java) and data (XML). This synergy is one of the core goals of strong application development. We're seeing this shift already with the rapid popularity of XML and the progress of the ebXML specification, which is an open, XML-based infrastructure for electronic business and international trade being developed through a joint effort of the United Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT) and the Organization for the Advancement of Structured Information Standards (OASIS).

**Should we expect a true component-oriented language to evolve? Who will create it?**

**Stroustrup:** I suspect the reason for a lack of success in this area is that people—primarily nonprogrammers—expect too much from a vague notion of "components." These people dream that components will somehow make programmers unnecessary. Instead of lots of unpredictable geeks writing code, neatly suited "designers" would compose systems out of prefabricated components using drag-and-drop. For tool vendors, the added attraction to this scenario is that only they would retain the skills needed to write those components.

The fundamental fallacy of this vision is that it is extraordinarily hard to design and implement components with a wide appeal. A single component or framework that does most of what is needed for an application or an industry would be attractive to its owner, and isn't technically too hard to build. But the various players in that industry would soon realize that if everybody used those components, they'd have no good way to differentiate their products from those of their competitors. They would become purveyors of a commodity, and the main profits would go to the component/framework supplier.

Tiny "components" can be useful, but don't offer much leverage. Medium-sized, more general components can be very useful, but such components require great flexibility in their composition.

In C++, we have made some progress by using combinations of class hierarchies for various forms of sharing and templates for tailoring interfaces. I expect something interesting and useful to emerge in this area, but I suspect it is more likely to be a new style of C++ use than a new language.

**Lindholm:** Writing component-oriented software seems to be more about adequate investment, good design, and programmer discipline than about programming languages. Certainly some languages better support writing such software than others.
But it would be misguided to expect that a magic new language will make writing components vastly easier than with current languages.

**With Microsoft putting its full weight behind C#, where does this leave other languages?**

**Stroustrup:** C++ will remain a major language for decades to come. It will evolve to meet new challenges. Languages in which useful systems are built don't just fade away.

I expect Microsoft realizes it has a significant interest in C++ (that is, ISO Standard C++) as a bridge to the rest of the computing world, as well as a tool for builders of large systems, for people with high-performance needs, and for embedded systems. The alternative languages seem aimed at middle-of-the-road business applications.

**The Competition**

**Will C# gain widespread acceptance—at the expense of other languages?**

**Lindholm:** Languages generally don't gain or lose at the expense of other languages. The hard-core Fortran programmer still programs in Fortran. While individuals might certainly shift their language of choice over time, languages are often additive items—meaning they exist "as well as" rather than "instead of."

Acceptance of a new language is generally tied to what it enables. Java technology's acceptance is driven by a combination of factors, including the Internet and the World Wide Web interface, frustration with previous technologies, and the ability to influence all aspects of Java technology's developmental direction. Another critical reason for Java's acceptance has been vendor independence, that is, choice of sources for compatible products.

Will C# gain widespread acceptance? It depends. Speculation is certainly an inexact science. In general, though, developers who aren't concerned with platform and vendor independence might like C#. For people tied to the Microsoft platform, it might also be a reasonable alternative to Visual Basic or Visual C++. But developers who are concerned with the ability to run the same application across different operating systems and environments—a key requirement for the distributed computing model with multiple access devices—will stay with languages, such as Java, that provide a standard, vendor-neutral runtime environment.

**Stroustrup:** The acceptance of C# will depend almost exclusively on the amount of money that Microsoft will spend on it.

It seems obvious that any significant use of C# would have to be at the expense of some other languages. However, it might not be so. Java seems to have gained a significant following without causing a decrease in C++ use. C++ use still seems to be growing steadily (rather than explosively). Maybe there is simply room for another language.

However, I fail to see the need for yet another proprietary language. In particular, I wonder why Microsoft would want both VB and C#.
What are the strengths—and weaknesses—of the various OOP languages?

**Stroustrup:** C++’s advantages have always been flexibility, efficiency, and its nonproprietary nature. These days, the ISO C++ standard enhances that last aspect.

I consider C++’s efficiency advantage fundamental. It comes from a model of data and computation that is closer to the machine than what Java and C# offer. The real question is for which kinds of problems this efficiency advantage is significant. I conjecture that there are many such problems. What people expect from their computers has always outstripped even the most dramatic advances in hardware technologies. Clearly, the designers of Java and C# (correctly) estimate there are many problems where efficiency is less important.

C++’s major weaknesses come from poor teaching (from people who insist C++ is exclusively an OOPL and from people who insist it is a minor variant of C), from variations of support on different platforms, from implementations with incomplete ISO standard conformance, and from the paucity of platform-neutral, system-level libraries.

I think a major part of these problems comes from the fact that C++ doesn't have a dominant vendor or owner willing to pour millions into development of libraries with an appeal to the whole C++ community.

**Lindholm:** Java technology is successful because it appeared in the right place at the right time with the right set of choices about what was important for the language and a computing platform to support. It's not a case of Java being a universally superior language to other OOP languages; it just did a better job of solving new problems that had arisen. It was made to suit the context of computing for the Internet. It omits the obscure constructs of C++, it successfully navigates the thorny issues of inheritance. And the inclusion of garbage collection significantly enhances productivity and reduces complexity. The use of a virtual machine in the context of the network and the corresponding design choices about security and dynamic loading also satisfied an emerging need and desire. These features ensured that Java technology got added to the repertoire of many extant programmers and enabled a marketplace for the skills of new programmers to flourish.

Also, Java's classes have a standardized, binary form, which is necessary (but not sufficient) for platform and vendor independence. The need for platform and vendor independence means the technology must be well-specified and must omit features that would make a binary standard impossible. Despite C++’s being an ISO standard, it's not possible to codify a practical binary standard across versions of C++ even within the same system and instruction-set architecture.

Many historical programming languages implemented using virtual machines gained reputations for less-than-stellar performance due to naive, slow interpreters or disruptive garbage collectors. Java's early implementations were similarly subject to such criticisms. But since those days, enormous investments in new virtual machine implementation technologies have been made across the industry. These efforts have so radically improved Java's performance that it rivals that of conventional, statically compiled languages for most uses. Programmers thus gain the advantages of cross-platform and vendor-independent program deployment typically without performance cost.
C++ does not enforce an object-oriented approach, and requires a fair degree of discipline on the part of the programmer to produce good object-oriented code. Many companies have become disenchanted with C++ for this reason. One of the great appeals of Java as a language is that it enforces an object-oriented approach and doesn't allow non-object-oriented constructs.

C# falls somewhere between C++ and Java. It has one foot on the boat and one on the dock, and thus fails to be fully safe and is relatively complex.

**Adopting a new language is a very expensive endeavor for companies. You can't hire (nobody knows the language), training cost is high, learning curves cause productivity hits, years of experience are washed away, and so on. How can a language overcome these obstacles?**

**Lindholm:** As noted in this question, adoption of something new often comes at a cost. The real issue is: Does this new thing provide significant cost savings or other improvements that offset the cost of switching? Companies have found that switching to Java technology has provided significant cost savings on the back end of development (time to market, quick iterations, easy maintenance) as well as the front end (ability to deploy to different operating systems, range of the technology’s strengths from devices to high-end servers, security).

Adoption of something new is also often forced by pain. To a large extent, this was the case with Java, which was developed largely in response to the inappropriateness of then-existing systems. Java technology relieved developer pain by: 1) emerging at the right place and the right time with respect to networking; 2) applying good taste and common sense in feature and capability selection; and 3) doing all these things in a reasonably compelling package. Java technology is also taught at the majority of universities—this has greatly helped keep the developer pool growing.

But most important, no other programming technology gave programmers the ability to develop applications for the Internet's heterogeneous environment. The evidence that the Java platform has done these things is overwhelming—a relatively short lifetime with a huge number of programmers and projects and capabilities demonstrated and underway, all of which are steadily growing. Java has become the default application platform for the Internet, and Java APIs are the natural platform for Internet application developers to write to.

**Stroustrup:** Microsoft and Sun are not plowing money into Java, Visual Basic, and C# out of the goodness of their hearts or because they believe it will make programmers' lives better. They do it to increase their own profits.

I have little doubt about the economic analysis behind that: Tool vendors can gain a significant economic advantage by diverting revenue and expertise from application builders. This is much harder to do if the application builders use open, standardized tools so they can rely on multiple sources for their needs.

**C++ has had genericity (that is, templates and STL), operator overloading, and enum types for a number of years. Will we see these in future releases of Java? Should we?**

**Stroustrup:** C++ has had templates since 1988-89. However, it's taken time to determine how best to use them, and early support varied in quality. Some compiler vendors were slow to provide support and at least one major vendor engaged in
disreputable anti-template propaganda until that vendor was able to support templates. Even today, the quality of template support tends to be patchy.

I think that Java (and C#) could and should be improved in the ways mentioned in this question. This would provide a real convergence of features and concepts in a way that would benefit programmers. That said, I think facilities such as destructors and template specialization are far more important than enumerations.

**Lindholm:** Java technology's success is in part a consequence of what it doesn't do. The question that needs to be asked is, "Is the feature essential, and what is the cost of adding it to the language?" Operator overloading in C++ is in some ways a strong feature, but it also adds a great deal of complexity to the language and is difficult for many people to deal with. By making an inspired set of choices among the various tradeoffs that could be made, Java technology hits a sweet spot of capability and need.

Certainly Java will evolve, and most importantly, the developer community now drives this evolution. The addition of genericity is a great example of how Java will evolve through the community's involvement in determining the correct balance between tradeoffs. A Java Specification Request (JSR) for adding generic types to the Java technology has been submitted to the Java Community Process (JCP) program and is far along in development (see info on JSR-014 at [http://java.sun.com/aboutJava/communityprocess/](http://java.sun.com/aboutJava/communityprocess/)). More than 80 JSRs are actively in development within the JCP, illustrating a significant developer response and acceptance of a community process that is driving the evolution of the Java platform.

**Evolution vs. Revolution**

C++ is an evolving language; both C# and Java can be seen as revolutionary (new from the ground up) languages. When are revolutionary programming languages necessary?

**Lindholm:** Java technology isn't revolutionary in itself, but rather evolutionary: Almost all of its features have existed in at least one other environment prior to the emergence of the Java platform. Java technology's valuable contribution is the choice of the set of features and the tradeoffs that have been made in making those choices to produce something both practical and aesthetically satisfying. Java technology cradles—but does not coddle—the programmer.

**Stroustrup:** I don't see Java or C# as being technically revolutionary or "from the ground up." Had Java first been designed from technical principles, it wouldn't have had the ugly and illogical C/C++ syntax.

I think a radical new language is needed only when there are radical changes in the problems faced by programmers or if we discover radically better programming techniques that are not supported by existing languages. The snag is that we'll never reach consensus on what "radical" means.

I think the only radically new programming techniques since the emergence of OOP are the generic programming and generative programming techniques that primarily originate from use of C++ templates—and even those could be seen as simply taking techniques with a long history in object-oriented and functional languages and making them formal, accessible, and affordable. However, I'm excited about what is currently being done with template programming in C++. For example, libraries such as POOMA,
Blitz++, and MTL have changed the way numerical computation is done in many places.

**One of Java's—and C#'s—claimed benefits is simplicity. Is Java losing this advantage?**

**Stroustrup:** New languages always proclaim their simplicity and denounce the needless complexities of older languages. Much of that "simplicity" is simply immaturity. A language's complexity comes from dealing with the rather nasty and varied complexities of the real world. If a language survives for long enough, complexity creeps in somewhere: in the language itself, in libraries, or in tools. C++ and Java are not exceptions to this rule; neither is C#. If it survives infancy, it too will grow dramatically in size and complexity.

**Lindholm:** The functionality included in Java technology has increased, and with that there's more to learn. But increased functionality hasn't come at the cost of increased complexity—the evolution of the Java technology hasn't changed the slope of the learning curve; it's just extended the curve further to the right.

**Standards**

**What are the advantages/disadvantages of a standardized, open language?**

**Lindholm:** There are no disadvantages to a language—or a runtime environment—that is truly open and that implements a recognized, enforceable standard as long as it can't be tainted by proprietary extensions, which equal vendor lock-in. It's also critical that the full platform be standardized, not just a subset, to avoid vendor lock-in with proprietary higher-level APIs. Customers now require the flexibility of vendor choice. They need both innovation and compatibility.

**Stroustrup:** The key advantage of a formal standard, such as the ISO standards for C and C++, is that it makes it hard for a vendor to manipulate the language for its own commercial gain. It provides for multiple vendors, which lowers prices for users and provides long-term stability.

The advantage of proprietary languages is marketing-fueled popularity, cheap facilities (until the users have become locked in), and rapid responses to perceived commercial needs and fads.

An important aspect of a formally standardized language is that users with unusual needs cannot be ignored. For example, working for AT&T, I have concerns for scale, reliability, and efficiency that far outstrip what mass-market vendors focus on. A company with mass-market focus naturally focuses on the needs of the majority.

However, most larger organizations and most cutting-edge companies have unusual needs. C++ was designed to be open-ended, flexible, and efficient, to be able to serve in situations I couldn't possibly have imagined. This is why C++ is far less paternalistic than other modern languages. People who disagree philosophically deem this "too dangerous."

Formal/open standards primarily serve the users/consumers of programming tools. Proprietary "standards" primarily serve the vendors.
About the Authors:
Bjarne Stroustrup is the designer and original implementor of C++, and the author of The C++ Programming Language and The Design and Evolution of C++. A graduate of the University of Aarhus, Denmark, and Cambridge University, England, Dr. Stroustrup is the head of AT&T Labs’ Large-Scale Programming Research Department, an AT&T Fellow, an AT&T Bell Laboratories Fellow, and an ACM fellow.

Tim Lindholm is a distinguished engineer at Sun Microsystems Inc. He was an original member of the Java platform development team at Sun, and was responsible for Java’s threading, synchronization, garbage collection, and memory management. He is a coauthor of The Java Virtual Machine Specification, and is currently working on platform architectures across the Java technologies.

Interview With Anders Hejlsberg

Anders Hejlsberg, distinguished engineer for Microsoft, is the driving force behind C#, the new OOP language to be included in the upcoming Visual Studio.NET. The editors at Fawcette Technical Publications met with Hejlsberg at the July 2000 Microsoft Professional Developers Conference to discuss the implications of C#—what's different about it, what's the same, who Microsoft expects to use it, and what it means to the future of software development.

What's wrong with C++ that Microsoft felt it needed to fix with C#?

First of all, C++ isn't going away. But the amount of power and control that C++ gives you sometimes gets in your way. There's clearly a need for simplification; literally every C++ user has expressed this over time.

Also, C# takes advantage of the common-language runtime. We're putting in place a whole bunch of core services in the system—garbage collection, exception handling, type safety, base-class libraries. It's hard to sprinkle garbage collection on after the fact. Designing a new language from scratch with a preservation of C++ heritage gives you the best of both worlds.

There are so many assumptions in C++ about pointers and operators and so forth, that you can't just do away with it and expect the code to exist or still continue to run. The world is moving toward component-style programming, where your application is gradually getting spread out into a bunch of smaller components that host in various environments. It's important to factor that into your language design and include all the concepts of component programming—properties, methods, events, attributes, documentation—and enable programmers to do them all in one place.

If you look at C++, you must do a lot of housekeeping—.h files and .idl files—and your code is spread out. It has a declaration here and an implementation there, and you have to do all this manual synchronization. Not only does that make it more
complicated, but it also makes it hard for the language to embed itself in other hosting environments. Embedding C++ in an ASP page is just not possible. What will you do? How do you include an .h file in there?

**What parts of C++ did you think were essential to keep in C#?**

Obviously, object orientation is key; we definitely still have that. The bulk of a program consists of ifs, whiles, do, fors, and so forth. Why mess with that? So, we kept the statement structure. C# will feel comfortable to C++ developers—the types have the same names and so forth. But then we did away with all the things that irritate programmers, such as: Why are there three different kinds of deference operators in C++?

**Who will have an easier time learning C#? C++ developers or Java developers?**

Both will have an easy time of it. Clearly, they're all in the same family of names—the curly-brace family, if you will. C++ programmers will definitely have an easier time going to C# than they would going to Java. Java made a lot of sacrifices from C++ that I find surprising. Take one example: Why are there no enums? An enum is a useful concept; it gives you type constants. For some reason, they're not there.

We preserved a lot more things: namespaces, operator overloading. Then, when you combine those with the type system unification we've done—where everything is viewed as an object—the kind of simplicity, extensibility, and usability you get is quite remarkable. In-house, we have several million lines of C# code now in the platform. Our programmers consistently report that it's a short learning curve from C++, and they find themselves way more productive.

**Microsoft is certainly putting its weight behind C#. Do you think there's going to be a trend away from C++?**

Both have their place. C++ is not going to go away. There's just no way—there's too much code out there. I would never propose to people that they throw away all the code they've already written. I actually view it differently, say, from the Java design point. Interoperability is key. You've just got to leverage what's there. That's what software engineering is becoming more and more about.

**Apart from maintaining your legacy code, are there circumstances where developers should be writing in C#, and others where they should be writing new code in C++?**

If you look at what we're doing with the common-language runtime and base-class libraries, it used to be that your choice of programming language also dictated your choice of API and your choice of program involved. We've eliminated all that. Now your programming language is a lifestyle choice. What are you more comfortable programming in? Regardless, you will have all the benefits of the runtime, all the same class libraries available.

I've been asked by many VB programmers, "should I switch to C#?" I tell them, no, you don't have to. That said, however, C# is a great language. You can always be cleaner in your design when you start from a blank slate, and obviously there is no
backwards compatibility baggage we’ve had to carry forward. Maybe our type names are a little more consistent and a little more in tune with the 32-bit world—INTs are 32 bits and not 16. If you have code in C++ you want to move into the .NET framework, Managed C++ is the right language. If you're starting from scratch and you're going to write for this new Web services platform, C# is probably your better choice.

What is the role of XML in C#?

It's hard to say, is this a language feature or a framework feature? We have great XML support in the .NET framework that's miles ahead of the competition—I think we're way more standards-compliant than anybody else. But even in the language, we have this new concept—which I think is truly innovative—called Attributes. Attributes allow you to dream up new things you can declare in the language. When you have fields and methods and properties, you want to put contextual, declarative information on them. Just as you can say whether they're public or private, you want to be able to say, in XML, does this become an element or an attribute? What is the root namespace for this class when it's put into an XML document?

We could have taken the traditional approach, adding new pragmas and keywords and making the language closed-end—we can go there, but our users can't. Instead, we took a step back and invented a mechanism that allows you to add new declarative information in the language and spread it out across domains. So, the mechanism we're using in the Web services infrastructure to describe the mapping from a class to an XSD schema and to XML, is the same mechanism we use for security to describe permissions. It's the same methodology we use for interoperability with a platform, to describe the DLL entry-point name of a function.

What's cool about C#: It's totally extensible. We build platforms where the ultimate end-users plug in components. And, of course, they will all have some contextual information they'll want their components to provide. If you're writing, say, a two-phase commit transaction manager, you're going to want some sort of contextual information to describe the business objects that plug into it, and this is precisely the mechanism; it's all type-safe, extensible, and multilingual.

VB developers might not care so much about C# or have any animosity toward it, but perhaps some C++ developers will say, "Hey, wait, you're messing with my religion." Do you anticipate any type of severe resistance from C++ developers?

It's human nature to resist change, first of all. And I am not religious about the choice of language—you pick the right tool for the job. Key to our message here: It's more of a lifestyle choice at this point. There's a multilingual platform there—you can switch between VB, C#, or C++—whatever works best for you, whatever leverages your skills more.

In your design of the language, did you anticipate the possibility of implementations on other platforms?

Yes, I would say so. Nothing in the language is Windows-specific. A lot of FUD has been floating around—everything is a COM object, everything is this and that—that's just not true. In fact, even our extensibility mechanism—the mechanism whereby we describe how a class in C# becomes a COM object, and all the plumbing you have to
provide—is all done through Attributes. And Attributes is a generic mechanism that's built into the language.

**So you could do a CORBA implementation, or even a Java one?**

Because the Java platform was built for one language—and literally all the way down the stack—that platform reflects choices made in that language. Certain features are missing in the Java VM, such as the notion of user-definable value types. Java has primitive types, and then there are objects, and the objects are always evaluated. We also have the notion of value types or structs in C# that are stack allocated—a decimal, a date time, a SQL datatype, a point, or a rectangle—small structures where it seems excessive to allocate them dynamically.

I don't know if you've ever used the big-decimal type in Java. Whenever you're writing a financial application in Java, you need more precision than a float or a double. The solution available is this thing called big-decimal; it's an object that has a method .add. So to add two things, you use total.add whatever as opposed to a plus operator. And, by the way, remember to check whether that other number was null—not that it at all makes sense for a number to be null.

We've taken a different approach and given you the ability to implement value types, as we call them, or structs, and implement operators on those things so they look like they're completely natural. But, of course, that does require some infrastructure support, and that's not present in the Java VM. The Java VM doesn't support attributes or unsigned types either. It just speaks to the fact that we built a multilingual platform and, therefore, implemented the grand union of commonly used language features as opposed to the features of one language.

---

**Glossary**

**Generic programming:** The ability to define abstract objects that are specified only by their interfaces and behaviors. A language that supports generic programming allows the definition of types you can specialize by providing parameters to the type that adapt the abstract behavior to a particular instance (for example, collections of integers and collections of strings should behave the same at some abstract level).

**EJB:** Enterprise JavaBeans, Java's distributed object technology.

**JSP:** Java Server Pages, Java's server-side scripting environment for serving up Web applications.

**Inheritance:** A programming language's support for objects' ability to use interfaces or implementations from other objects without requiring you to rewrite the code. This
allows you to extend an object's capabilities by creating a new object that extends the behavior of the base object.

**Garbage collection:** Automatic memory management performed by the runtime environment. In languages that have garbage collection, the programmer doesn't need to explicitly release objects and resources; the system will release them when it determines the program no longer needs them.

**Heterogeneous environment:** A computing environment where multiple hardware platforms, operating systems, and programming languages can coexist and interact.

**Genericity:** the extent to which a language supports generic programming.

**Overloading:** Specifying more than one version of an interface method with the same name but a different signature (different parameters, and/or potentially a different return type). For example, you could have a Count() function that could take either an array of strings or a reference to a collection object.

**Destructors:** Object methods that destroy or release object resources right before the objects themselves are destroyed. This allows objects to be self-managing, in that they can release resources they've obtained for cleanup, or save their state to some more permanent resource for future use.

**Template specialization:** The ability to extend the capabilities of templates without having to duplicate the code. This is a similar concept to implementation inheritance for objects, applied to templates.

**Generative programming:** The software development process that supports the development of generic types. This encompasses not only generic programming itself, but the software development processes of modeling and specification that allow you to quickly adapt generic programs to a specific application or requirement.

**Namespaces:** A language construct that allows programmers to specify which parts of an application a particular declaration is available in. A declaration in one namespace can have the same name as a declaration in another namespace, without clashing.